Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Septiembre-Diciembre 2010

Carnet:_____

Nombre:_____

Examen III

(Versión A – 40 puntos)

Pregunta 0	Pregunta 1	Pregunta 2	Total
20 puntos	10 puntos	10 puntos	40 puntos

Pregunta 0 - 20 puntos

Esta pregunta consta de diez (10) subpreguntas de selección, numeradas de 0.0 a 0.9. Cada una de las subpreguntas viene acompañada de cuatro posibles respuestas (a, b, c, d), entre las cuales sólo **una** es correcta. Ud. deberá marcar completamente y sin posibilidad de confusión la opción que considere correcta o, si desea no contestar la subpregunta, no marcar ninguna de las opciones, si Ud. marca más de una opción su respuesta será considerada omitida.
Cada una de las subpreguntas tiene un valor de dos (2) puntos. Tres subpreguntas incorrectas eliminan una correcta. Las subpreguntas omitidas (en las que se han dejado todas las opciones sin marcar o se han marcado más de una opción) no suman ni restan puntos.

0.0. Considere el siguiente programa escrito en pseudocódigo:

```
var n := 10 : int
    m := 10 : int

    proc p (y : int, z : int)
    begin
        y := m + y
        z := 5 + n
    end

begin
    p (n, n)
    escribir(n)
end
```

en el cual `escribir` es una subrutina que escribe en pantalla el valor pasado como argumento, considere que el pasaje de parámetros se realiza por valor en el caso de y, y por valor / resultado en el caso de z. ¿Cuál es el valor que se muestra en pantalla?

a) 10

b) **15**

c) 25

d) Ninguno de los anteriores

0.1. Continúe con el programa dado en la subpregunta (0.0), pero ahora considerando que el parámetro y es pasado por valor / resultado, y el parámetro z es pasado por referencia, ¿cuál es valor que se muestra en pantalla?

a) 10

b) 15

c) 25

d) **Ninguno de los anteriores** *(20)*

0.2. Considere que tiene las clases A, B y C declaradas en C++ como se muestra a continuación:

```
class A {
   T x;
   T y;
   public:
      virtual void m() = 0;
}
class B : public A {
   T z;
   public:
      virtual void p () { ... }
}
class C : public B {
   T w;
   public:
      virtual void m () { ... }
}
```

y se tienen las siguientes declaraciones:

```
A* a0;        // línea 0
A a1;         // línea 1
B* b0;        // línea 2
B b1;         // línea 3
C* c0;        // línea 4
C c1;         // línea 5
```

¿cuáles líneas corresponden a declaraciones no válidas?

a) 1, 3 y 5.

b) **1 y 3.** *(Clases abstractas no se pueden instanciar implícitamente en el modelo de valor)*

c) 2 y 4.

d) Ninguna de las anteriores.

0.3. Continúe considerando las clases A, B y C, mencionadas en la pregunta 0.2., y considere el siguiente fragmento de código:

```
A* a = ...;    // inicialización apropiada de a
B* b = ...;    // inicialización apropiada de b
C* c = ...;    // inicialización apropiada de c
a = b;         // asignación i)
b = a;         // asignación ii)
a = c;         // asignación iii)
c = a;         // asignación iv)
```

¿cuáles de las asignaciones son semánticamente correctas en cualquier caso?

a) **i) y iii)** *(Apuntador a objeto de una clase puede apuntar a objetos de subclases)*

b) ii) y iv)

c) Todas.

d) Ninguna.

0.4. Continúe considerando las clases A, B y C, mencionadas en la pregunta 0.2., y el fragmento de código de la pregunta 0.3. ¿cuáles de esas asignaciones serían semánticamente correctas si se agrega una conversión explícita?

a) i) y iii)

b) **ii) y iv)**

c) Todas.

d) Ninguna.

0.5. Continúe considerando las clases A, B, y C, mencionadas en la pregunta 0.2., y considere el objeto asociado a la variable c en la pregunta 0.3. y que éste está almacenado en la dirección de memoria 5000, ¿en qué dirección se encuentra el método `m` que será efectivamente ejecutado cuando se haga la llamada: `c->m()`? (considere que el tamaño de un apuntador y del tipo T es X)

a) `*(*5000)`. *(El método `m` es el primero en el v-table para la clase, pues se define en A)*

b) *(*5000 + X).

c) *(*5000 + 2*X).

d) Ninguna de los anteriores.

0.6. Considere la siguiente función en Scheme:

```
(define x
  (lambda (L)
    (cond
      ((null? L) L)
      ((null? (cdr L)) L)
      ((eqv? (car L) (car (cdr L))) (x (cdr L)))
      (else (cons (car L) (x (cdr L)))))))
```

y la siguiente llamada:

```
(x '(1 2 2 3 3 3 4 4 4 4 5 5 5 5 5))
```

¿cuál es la lista devuelta por esta llamada?

a) **(1 2 3 4 5).**

b) (1 2 3 3 4 4 4 5 5 5 5).

c) (2 3 3 4 4 4 5 5 5 5).

d) Ninguna de las anteriores.

0.7. Considere el siguiente procedimiento en C++:

```
template<class T>
void p(T A[], int A_size) {
   ...
}
```

¿Cuál de las siguientes afirmaciones es cierta?

a) **En el cuerpo de este procedimiento se puede usar cualquier operación que sea común a todas las clases del lenguaje, y el compilador debe emitir un mensaje de error en caso de usar sobre A una operación que no sea ofrecida por todas las clases del lenguaje.**

b) En el cuerpo de este procedimiento se puede usar cualquier operación ofrecida por las clases que tengan el método `operator[]`, y si se usa con un arreglo de objetos de una clase que no lo ofrezca, el comportamiento es impredecible.

c) Se debe hacer una instanciación de este procedimiento por cada clase con la cual se quiere utilizar, y al realizar la instanciación se debe proveer las operaciones necesarias a usar en el cuerpo del procedimiento.

d) Ninguna de las anteriores

0.8. ¿Cuál de las siguientes afirmaciones es cierta con respecto al manejo de excepciones en un lenguaje como C++?

a) Es irrelevante mantener la pila en un estado consistente, la excepción provocará que el programa aborte su ejecución.

b) **Es relevante mantener la pila en un estado consistente por ello será necesario que el llamado incluya un manejador de excepciones implicito que se encargue de realizar el epílogo cuando la excepción sea propagada al llamador.**

c) Es relevante mantener la pila en un estado consistente por ello será necesario que el llamador se encargue de desempilar el registro de activación del llamado cuando la excepción le sea pasada.

d) Ninguna de las anteriores.

0.9. Considere el siguiente predicado en Prolog:

```
misterio(A, B, C) :- A = B, C = A.
misterio(A, B, C) :- A > B, D is A - B, misterio(D, B, C).
misterio(A, B, C) :- B > A, D is B - A, misterio(D, A, C).
```

y las siguientes consultas:

```
? - misterio(2, 2, 2).
? - misterio(6, 4, 1).
? - misterio(8, 9, 6).
? - misterio(5, 25, 5).
```

¿cuáles son las respuestas dadas para estas consultas?

a) Yes, Yes, Yes, Yes.

b) Yes, No, Yes, No.

c) No, Yes, No, Yes.

d) **Ninguna de las anteriores.** *(Yes, No, No, Yes)*

Pregunta 1 - 10 puntos

Algunas implantaciones de Prolog ofrecen un predicado trace/0 que activa un mecanismo de traza en la evaluación. El mecanismo hace que cada vez que se inicia un proceso de resolución, se muestra 'Call: ' y el objetivo a probar. Si el objetivo es probado, se muestra 'Exit: ' y el objetivo resultante. Si el objetivo no puede ser probado, se muestra 'Fail: ' y el objetivo original. Por último, si se solicitan más soluciones para un objetivo, se muestra 'Redo: ' y el objetivo a probar nuevamente.

Si su implantación de Prolog no cuenta con el predicado trace, Ud. siempre puede implantar un meta-evaluador que provea el comportamiento, incluyendo la organización cosmética de indentar las llamadas anidadas para hacer evidente la profundidad de la recursión. Así, el uso de este evaluador sería similar a:

```
?- probar(member(X, [a,b,c])).
Call: member(_G212, [a, b, c])
Exit: member(a, [a, b, c])
X = a ;
Redo: member(a, [a, b, c])
 Call: member(_G212, [b, c])
 Exit: member(b, [b, c])
Exit: member(b, [a, b, c])
X = b ;
Redo: member(b, [a, b, c])
 Redo: member(b, [b, c])
  Call: member(_G212, [c])
  Exit: member(c, [c])
 Exit: member(c, [b, c])
Exit: member(c, [a, b, c])
X = c ;
Redo: member(c, [a, b, c])
 Redo: member(c, [b, c])
  Redo: member(c, [c])
   Call: member(_G212, [])
   Fail: member(_G212, [])
  Fail: member(_G212, [c])
 Fail: member(_G212, [b, c])
Fail: member(_G212, [a, b, c])
false.
```

Considere el meta-evaluador básico para Prolog estudiado en clase y que imita el comportamiento del evaluador natural de Prolog:

```
probar(true).
probar((Objetivo1,Objetivo2)) :-
    probar(Objetivo1),
    probar(Objetivo2).
probar(Objetivo) :-
    clause(Objetivo,Cuerpo),
    probar(Cuerpo).
```

Modifique el meta-evaluador `probar/1` de tal manera que se muestre la traza de ejecución anidando los niveles de correspondientes a llamadas recursivas de la forma previamente descrita.

Puede utilizar cualquiera de los predicados estándar de Prolog, pero implantando cualquier otro predicado auxiliar que considere necesario.

Las modificaciones necesarias al predicado modelo son simples:

1. Escribir un predicado auxiliar que reciba un argumento adicional que representa la indentación necesaria.
2. Garantizar, vía *green cuts*, que cada camino de demostración **exitosa** se toma una sola vez.
3. Garantizar que después de un `Exit`, se deje la posibilidad de hacer `Redo` aprovechando el *backtracking*. Para ello, el primer `Redo` tirunfa trivialmente con `true` sin emitir diagnóstico, pero el segundo fuerza una falla indicando el proceso de `Redo`.
4. Para ahorrar trabajo, se dispone de un predicado para la mayoría de los diagnósticos.

```
prove(Goal) :- traceit(Goal,0).

traceit(true, _)   :- !.
traceit((G1,G2),P) :- !,
  traceit(G1,P),
  traceit(G2,P).
traceit(G,P)       :-
  msg('Call: ',G,P),
  clause(G,B),
  P1 is P+1,
  traceit(B,P1),
  msg('Exit: ',G,P),
  redo(G,P).
traceit(G,P) :-
  msg('Fail: ',G,P),
  fail.

% El primer redo/2 siempre triunfa silenciosamente, pues corresponde
% al Exit. Si el usuario solicita más soluciones, lo primero que
% ha de fallar es redo/2, y la siguiente alternativa anuncia que
% se intentará un Redo y se obliga a la falla, para que el backtracking
% nos llege hasta la llamada más reciente a traceit/2.
redo(_,_).
redo(G,P) :- msg('Redo: ',G,P), fail.

% Emitir el mensaje M para el objetivo G indentando P espacios.
msg(M,G,P) :- tab(P), write(M), write(G), nl.
```

Pregunta 2 - 10 puntos

En Scheme, las funciones `let*` y `let` permiten definir un alcance estático para la evaluación. La diferencia entre ambas es la forma en que se hacen las asociaciones: la primera las hace en secuencia y la segunda las hace simultáneamente. Esas funciones no son más que *syntactic sugar* pues son equivalentes a la aplicación funcional curryficada o no. Esto es, todo `let*` o `let` puede convertirse a una secuencia apropiada de `lambda`s.

Estamos implantando un interpretador de Scheme en Haskell, y definimos el siguiente tipo de datos para representar S-expresiones:

```
data S  = Lambda [String] S [S]     -- LAMBDA abstracciones (aplicacion funcional)
        | Expr [S]                  -- S-expresion simple
        | LetPar [(String,S)] S     -- LET en simultáneo
        | LetSeq [(String,S)] S     -- LET en secuencia
        deriving (Eq,Show)
```

El interpretador solamente opera con λ-abstracciones y con las expresiones simples de aplicación funcional, por lo que es necesario transformar todas las asociaciones simultáneas y en secuencia, para convertirlas en las λ-abstracciones adecuadas. Para esto, Ud. debe escribir la función

```
let2lambda :: S -> S
```

que recibe cualquier S-expresión y efectúa la transformación.

Nota: Ud. recibirá dos (2) puntos adicionales si construye `let2lambda` empleando *exclusivamente* funciones de orden superior.

Del enunciado se deduce que la función `let2lamba` solamente debe retornar S-expresiones de la forma `Lambda` o `Expr`.

Para los casos base, la función debe reconstruir la S-expresión `Lambda` y `Expr`, previo procesamiento recursivo de las S-expresiones anidadas. En el caso de un `Expr`, deben procesarse todas las S-expresiones en la lista; para el caso de un `Lambda`, deben procesarse ls S-expresión que conforma el cuerpo de la función y las S-expresiones que corresponden a los parámetros actuales de la aplicación funcional.

La forma especial `let` de Scheme, representada por el caso `LetPar`, corresponde a la aplicación funcional no curryficada. Por tanto, debe ser reemplazada por un `Lambda` cuyos parámetros formales sean todos los identificadores que aparecen en la lista de asociaciones, cuyo cuerpo corresponda a la S-expresión original procesada recursivamente con `let2lambda`, y aplicada sobre la lista de argumentos que corresponde a las S-expresiones en la lista de asociación, todas ellas procesadas recursivamente con `let2lambda`. Estos son dos casos típicos de aplicación de `map`.

La forma especial `let*` de Scheme, representada por el caso `LetSeq`, corresponde a la aplicación funcional curryficada. Por tanto, debe ser reemplazada por una secuencia de `Lambda` anidadas. La más interna debe utilizar el último identificador de la lista de asociaciones, la S-expresión original procesada recursivamente con `let2lambda` y aplicada sobre la última S-expresión de la lista de asociaciones, procesada recursivamente con `let2lambda`. Sabiendo que la más interna ya no contiene `LetPar` ni `LetSeq`, se procede recursivamente envolviendo cada `Lambda` así generada por otra `Lambda` que tiene como parámetro formal el siguiente identificador (de derecha a izquierda) de la lista de asociaciones y como parámetro actual la S-expresión correspondiente en la lista de asociaciones, procesada recursivamente con `let2lambda`. Este es un caso típico de aplicación de `foldr`: en cada paso, se recibe una asociación `(i,s)` y la S-expresión más interna, generando el `Lambda` correspondiente.

```
data S  = Lambda [String] S [S]
        | LetPar [(String,S)] S
        | LetSeq [(String,S)] S | Expr [S]
        deriving (Eq,Show)

let2lambda (Expr ss)        = Expr $ map let2lambda ss
let2lambda (Lambda is s as) = Lambda is (let2lambda s) (map let2lambda as)
let2lambda (LetPar bs s)    = Lambda is (let2lambda s) as
                                  where is = map fst bs
                                        as = map let2lambda $ map snd bs
let2lamdba (LetSeq bs s)    = foldr go (let2lambda s) bs
                                  where go b s' = Lambda [fst b]
                                                         s'
                                                         [let2lambda (snd b)]
```

Solamente muestro la solución utilizando funciones de orden superior, porque es realmente costoso (en espacio y en tiempo) escribirla utilizando recursión explícita.